```
// COS30008, List, Problem Set 3, 2021

#pragma once

#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"

#include <stdexcept>

using namespace std;
template<typename T>
class List
{
private:
       // auxiliary definition to simplify node usage
       using Node = DoublyLinkedList<T>;

       Node* fRoot;  // the first element in the list
       size_t fCount;       // number of elements in the list

public:
       // auxiliary definition to simplify iterator usage
    using Iterator = DoublyLinkedListIterator<T>;

       ~List()
    {
        while ( fRoot != nullptr )
        {
            if ( fRoot != &fRoot->getPrevious() )
            {
                Node* lTemp = const_cast<Node*>(&fRoot->getPrevious());

                lTemp->isolate();
                delete lTemp;
            }
            else
            {
                delete fRoot;
                break;
            }
        }
    }

    void remove( const T& aElement )
    {
        Node* lNode = fRoot;

        while ( lNode != nullptr )
        {
            if ( **lNode == aElement )
            {
                break;
            }

            if ( lNode != &fRoot->getPrevious() )
            {
                lNode = const_cast<Node*>(&lNode->getNext());
            }
            else
            {
                lNode = nullptr;
```

```
        }
    }

    // At this point we have either reached the end or found the node.
    if ( lNode != nullptr )
    {
        if ( fCount != 1 )
        {
            if ( lNode == fRoot )
            {
                fRoot = const_cast<Node*>(&fRoot->getNext());
            }
        }
        else
        {
            fRoot = nullptr;
        }

        lNode->isolate();
        delete lNode;
        fCount--;
    }
}

   // PS3 starts here

   // P1

List() :
    fRoot(nullptr),
    fCount(0) {}

bool isEmpty() const
{
    return fRoot == nullptr;
}

size_t size() const
{
    return fCount + 1;
}

void push_front(const T& aElement)
{
    if (isEmpty()) {
        fRoot = new Node(aElement);
        return;
    }
    Node* lNodeInsert = new Node(aElement);
    *fRoot->push_front(*lNodeInsert);
    fRoot = lNodeInsert;
    fCount++;
}

Iterator begin() const
{
    Iterator iter(fRoot);
    return iter.begin();
}

Iterator end() const
{
```

```
    Iterator iter(fRoot);
    return iter.end();
}

Iterator rbegin() const
{
    Iterator iter(fRoot);
    return iter.rbegin();
}

Iterator rend() const
{
    Iterator iter(fRoot);
    return iter.rend();
}

   // P2

void push_back(const T& aElement)
{
    if (isEmpty()) {
        fRoot = new Node(aElement);
        return;
    }
    Node* lNodeInsert = new Node(aElement);
    const_cast<Node*>(&fRoot->getPrevious())->push_back(*lNodeInsert);
    fCount++;
}

   // P3

const T& operator[](size_t aIndex) const
{
    if (aIndex > fCount)
        throw range_error("Index is out of range.");

    Node* lCurrentNode = fRoot;
    int lCount = 0;
    while (lCurrentNode != nullptr)
    {
        if (lCount == aIndex)
            return lCurrentNode->getPayload();
        lCount++;
        lCurrentNode = const_cast<Node*>(&lCurrentNode->getNext());
    }
}

   // P4
// copy constructor
List(const List& aOtherList) :
    fRoot(nullptr),
    fCount(0)
{
    for (size_t i = 0; i < aOtherList.size(); i++)
        push_back(aOtherList[i]);
}

List& operator=(const List& aOtherList)
{
    for (size_t i = 0; i < size(); i++)
        remove(operator[](i));
```

```
        for (size_t i = 0; i < aOtherList.size(); i++)
            push_back(aOtherList[i]);

        return *this;
    }

    // P5X

    // move features
    List(List&& aOtherList) :
        fRoot(nullptr),
        fCount(0)
    {
        for (size_t i = 0; i < aOtherList.size(); i++)
            push_back(aOtherList[i]);
    }

    List& operator=(List&& aOtherList)
    {
        operator=(aOtherList);
        return *this;
    }

    void push_front(T&& aElement)
    {
        push_front(aElement);
    }

    void push_back(T&& aElement)
    {
        push_back(aElement);
    }
};
```